(12)

Real-World Shepherding—Combining Vision, Manipulation, and Planning in Real Time

Peter von Kaenel and Robert W. Wisniewski

Technical Report 530
August 1994

UNIVERSITY OF
ROCHESTER
COMPUTER SCIENCE

Real-World Shepherding - Combining Vision, Manipulation, and Planning in Real Time

Peter von Kaenel and Robert W. Wisniewski
vonk,bob@cs.rochester.edu

The University of Rochester
Computer Science Department
Rochester, New York 14627

Technical Report 530

August 1994

Abstract

Designing real-world applications can involve coordinating many pieces of hardware and integrating multiple software components. Increased processing power has allowed complex real-world applications to be designed, and there has been increasing interest in the issues involved in designing both the applications and their support. In this paper we describe the issues involved in designing the application. The shepherding application we have chosen is representative of many real-world applications. This report focuses on technical details. We describe the underlying hardware, including the camera, vision processing boards, processors, and puma robot arm. We then discuss the software components we designed to integrate the hardware components in real-time. At each stage we describe the trade-offs between the different possibilities and why the ones chosen were best suited for our environment. We also present results supporting our selection. At appropriate points we indicate underlying support that would have eased and improved our implementation.

This material is based upon work supported by NSF Research Grant no. IRI-8903582 and CDA-8822724, DARPA Research Grant no. MDA972-92-J-1012, and ONR research grant no. N00014-93-1-0221. The Government has certain rights in this material. Robert Wisniewski was partially supported by an ARPA Fellowship in High Performance Computing administered by the Institute for Advance Computer Studies, University of Maryland.

REPORT DOCUMENTATION PAGE

Form Approved
OMB No. 0704-0188

1. AGENCY USE ONLY (Leave blank)	2. REPORT DATE	3. REPORT TYPE AND DATES COVERED
	August 1994	technical report

4. TITLE AND SUBTITLE

Real-World Shepherding—Combining Vision, Manipulation and Planning in Real Time

5. FUNDING NUMBERS

MDA972-92-J-1012, N00014-93-1-0221

6. AUTHOR(S)

Peter von Kaenel and Robert W. Wisniewski

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESSES

Computer Science Dept.
734 Computer Studies Bldg.
University of Rochester
Rochester NY 14627-0226

8. PERFORMING ORGANIZATION

9. SPONSORING / MONITORING AGENCY NAME(S) AND ADDRESSES(ES)

Office of Naval Research
Information Systems
Arlington VA 22217

ARPA
3701 N. Fairfax Drive
Arlington VA 22203

10. SPONSORING / MONITORING AGENCY REPORT NUMBER

TR 530

11. SUPPLEMENTARY NOTES

12a. DISTRIBUTION / AVAILABILITY STATEMENT

Distribution of this document is unlimited.

12b. DISTRIBUTION CODE

13. ABSTRACT (Maximum 200 words)

(see title page)

14. SUBJECT TERMS

real-world applications; SPARTAs; Ephor; runtime environments

15. NUMBER OF PAGES

25 pages

16. PRICE CODE

free to sponsors; else $2.00

17. SECURITY CLASSIFICATION OF REPORT	18. SECURITY CLASSIFICATION OF THIS PAGE	19. SECURITY CLASSIFICATION OF ABSTRACT	20. LIMITATION OF ABSTRACT
unclassified	unclassified	unclassified	UL

Contents

1 Introduction **2**

1.1 The Shepherding Application 2

1.2 Real-Time Aspects 4

1.3 Overview 4

2 Specifics of the Hardware and Software **4**

2.1 Maxware Code Modifications 4

2.2 Kernel Modifications 5

2.3 Digitization and Access 6

3 Decomposition and Real-Time Control **7**

3.1 User-Level Scheduler 8

3.2 Display Process 10

3.3 Communication Process 10

3.4 Planner Process 11

3.5 The Vision Process 12

3.6 The Manipulation Process 12

4 Vision **12**

4.1 Blobify 13

4.2 Associating Blobs to Objects 16

1) Associate Visible Blobs to Known Objects 17

2) Map Unassociated Blobs to Recently Unobscured Objects 18

3) Estimate Positions of Obscured Objects 18

4) Map Unassociated Blobs to New Objects 19

4.3 Filtering 19

5 Manipulation **22**

6 Conclusion **24**

1 Introduction

Designing our real-world shepherding application is part of a larger project of developing Ephor[1], a runtime supporting Soft PArallel Real-Time ApplicationS, or *SPARTAS* [4, 5]. In simulation we have been developing many mechanisms in Ephor designed both to increase the performance of SPARTAS and ease their implementation. Simulation provides the ability to try out many possibilities quickly and explore several different implementations or possible designs. The goal of Ephor is supporting real-world applications. Events not anticipated during simulation sometime occur in the real world and therefore may not be properly handled by a program designed solely on simulation. Thus, the goal of implementing the shepherding application in our robotics lab was to provide a real-world application to test and verify the mechanisms designed in Ephor based on simulation.

1.1 The Shepherding Application

It is important that the mechanisms developed in Ephor be generalizable to many applications. We chose the shepherding domain because it is flexible and maps onto a large class of real-world applications that involve real-time constraints and responsibilities, parallel hardware, dynamic resource management, uncertain actions, uncertain sensing, planning and replanning, dynamic focus of attention, and low-level reflexive behaviors, (e.g. purposive vision, autonomous vehicle control and navigation). The implementation runs in our robotics laboratory [1] and consists of self-propelled Lego vehicles "sheep" (see figures 1 and 2) that move around the table "field" (see figure 22) in straight lines but random directions. Each sheep moves at constant velocity until herded by the robot arm ("shepherd"), at which time it is redirected back towards the center of the field. The shepherd has a finite speed and can affect only one sheep at a time. Figure 3 illustrates the different hardware components involved in the shepherding application. The goal of the shepherd is to keep as many sheep on the table as possible; the more powerful the sheep behavior-models and look-ahead available to the planner, the better the results.

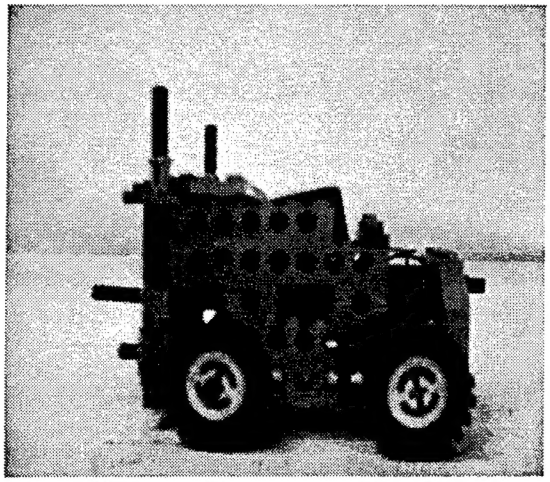

Figure 1: Side view of a Lego sheep.

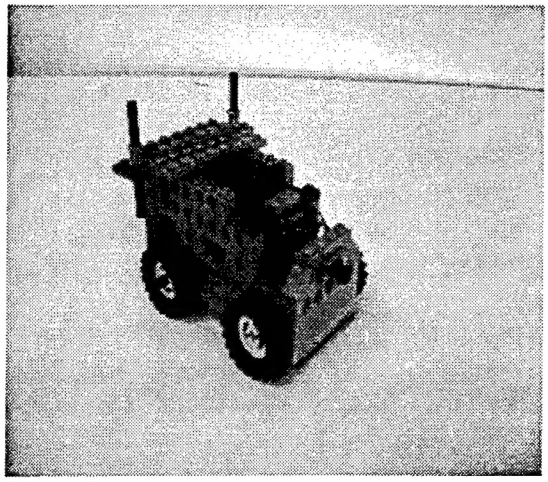

Figure 2: Top view of a Lego sheep.

The shepherding application is flexible and representative of a large class of applications. It includes high level cognitive models of the real world, planning, searching, sensing, acting, active perception, focus of attention, and multiple goals. It contains situations in which over demand

[1] Ephor was the name of the council of five in ancient Greece that effectively ran Sparta

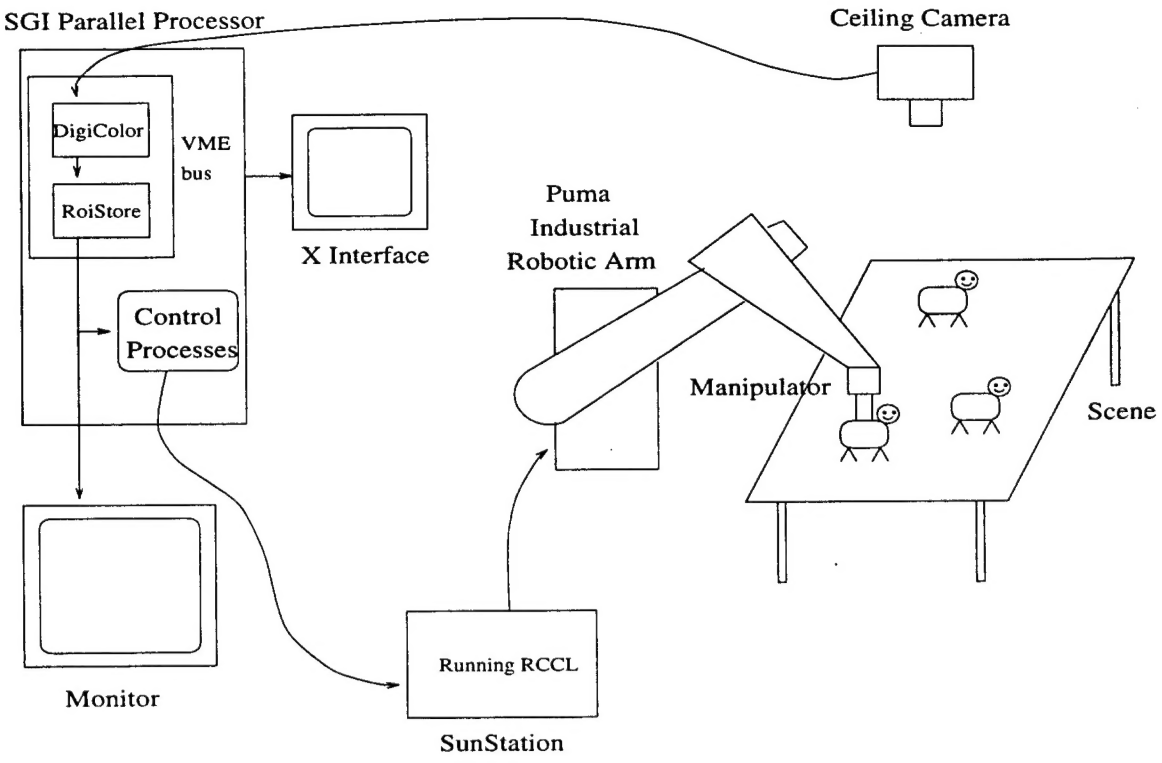

Figure 3: A flow diagram of the shepherding project.

can occur as well as the need for quick allocation and deallocation of resources. The shepherding application allows us to investigate many interesting properties of real-time systems that occur singly or in combination in other applications. Other real-world applications like navigation, game playing, laser tag, purposive vision, package delivery, and automated RSTA devices contain similar properties to the shepherding application. In varying degrees all contain an element of search whereby the agent determines the next course of action. Most are designed around a high-level executive instructing lower levels to carry out actions. The executive reasons using a model of the real world. Its requested actions are interpreted by lower level's modules and translated before being carried out. Many contain an intermediate layer responsible for small corrections to the requested action (servoing). Many also contain a low-level survival layer whose actions need to be carried out constantly and can occur "subconsciously", i.e., without intervention from the higher levels. The shepherding application includes all of these properties. It uses various search algorithms to determine the next sheep to save. The planning code is a high level executive working with models of the real world, sending out instructions to lower layers. The shepherding application's lower level interprets the requests from the executive (such as deflect sheep at x,y), and carries them out. An intermediate layer is employed to correct open loop instructions by the executive. In the shepherding application this may mean fine adjustments to ensure the manipulator actually deflects the sheep even if the specified time or coordinates are slightly off. The shepherding application also contains low level vision sensing that is constantly tracking the sheep. This occurs without the executive requesting it. Thus, the shepherding application embodies the important properties of many applications.

1.2 Real-Time Aspects

There are several crucial components involved in implementing the shepherding application: a real-time control component integrating the various modules, a vision module allowing tracking of the sheep, a communication module allowing the different machines to communicate across the ethernet, a manipulation module controlling the robot, and (for the user's convenience) a display module for debugging and tracking the application's progress.

The real-time control ensures each process runs at its specified intervals. In designing the modules to be used in the real-world shepherding special constraints needed to be satisfied. For example, often in the vision community separating an object from a scene could take several seconds per object, yet for real-time tracking we needed to track multiple objects many times a second. As another example, in the planning portions we needed to be able to place bounds on the time needed to make a decision. These examples provide insight into the difference between solving problems in real time versus off line. There are other requirements of the shepherding application, such as the need for high resolution object detection, that are discussed in detail in the appropriate sections.

1.3 Overview

Coordinating the different pieces of hardware with the developed software components requires an understanding of the requirements of both the underlying hardware and their associated libraries. Section 2 presents the details of the different hardware and associated software packages used to develop the real-world shepherding application. In sections 3, 4, and 5 we carefully discuss implementation and tradeoffs of our application. Specifically, section 3 presents the coordinating real-time component of the system and briefly describes the different processes and their functionality. Section 4 discusses the specific vision requirements needed by real-world shepherding, our solution to the problems, and the tradeoffs considered during implementation. We also discuss the algorithmic considerations of performing real-time multi-object object detection and tracking in section 4. Section 5 concludes the implementation section discussing the difficulties and solutions of performing real-time manipulation. We discuss the capabilities of the application and make concluding remarks in section 6.

2 Specifics of the Hardware and Software

This section contains detailed information about the hardware and software used to develop the shepherding application. The first two sections, 2.1 and 2.2, are specific to the the twelve processor SGI Challenge Series we attached the vision processing boards to and may be of interest only to readers with such platforms. The remaining section, 2.3, contains information on timings and board interconnections resulting from the configuration of Maxware boards we used.

2.1 Maxware Code Modifications

The Maxware software was originally written for a sun workstation and had to be ported to the mips based SGI multiprocessor running IRIX. Below are the difficulties encountered during this port and the solutions implemented. The solutions are the most straightforward to the problems that presented themselves and not necessarily the most elegant. It is also possible that some of the problems have or will cease to exist on future versions of IRIX, but these were the difficulties that arose when porting to IRIX 5.0.1.

The first set of problems encountered were the incompatibility of makefile formats. The makefile on the sun defaults to running */bin/sh*. IRIX defaults to running the shell of user executing the

makefile. Thus the line "SHELL = */bin/sh*" was added to the top-level makefile. On our version of IRIX, the default compile was "ansii", but the software was not written with these conventions in mind. To handle this difficulty, a "-cckr" was added to all "CFLAGS" options. In many of the makefiles the "OFILES" definition was defined using the "FUNCS" definition, which was a combination of "CFUNCS" and "OBJSUF". When the makefile program was combining and substituting other definition to form "OFILES", it was not obtaining the proper list of object files. To solve this, the "OFILES" definition was defined by individually listing out each object file. When the makefile invoked ranlib numerous errors resulted. There was an incompatibility between the formats of the object files and what ranlib was expecting. After unsuccessfully deciphering the message, we simply added a empty executable ranlib to the end of our $PATH variable thus bypassing the creation of the table of contents for the archive.

There we other errors requiring only minor changes:

- In the *dcLutInit.c* file in the *dc* subdirectory **for** loops with the start condition of "i=-128" were modified to "i=(-128)"
- In the *mvAlloc.c* and *mvPage.c* files in the *mv* subdirectory the following lines were added to correctly define the "PROT_EXEC" constant:
  "#ifdef sgi; #define PROT_EXEC PROT_EXECUTE; #endif"
- In the *mvControl.h* file in the *include* subdirectory the definition of "m" was changed to 'm'
- In the *msTool.c* file in the *mains/tools/maxsp* subdirectory the variable const was changed to _const_
- In the *Mdepends.bsd* file in the *dc* subdirectory the "$(FUNCS): $(INCS)" line was commented out
- In the *setecho.c*, *clrecho.c*, *setcbreak.c*, and *clrbreak.c* files in the *parser* subdirectory the initializing declaration of "static struct termio tty = 0;" was changed to the simple declaration of " static struct termio tty"
- In the *roiCalc.c* file in the *mains/roicalc* subdirectory an "#ifdef sgi" was added for the System dependent constant variables that matched the "#ifdef sun"

These changes were sufficient to compile all the code and successfully run the code for the DigiColor and ROIStore boards in "POLLED" mode.

2.2 Kernel Modifications

The Maxware coed was originally ported to the IRIX 4.0.3 operating system. This version of the operating system did not contain any VME drivers so a method had to be found that would allow access to the DigiColor and ROIStore boards on the VME bus. There was a *mem* file in the *uts/mips/master.d* subdirectory that defined an array of device addresses mappable by the */dev/mmem*. The addresses must be kernel virtual addresses, not physical addresses. We treated the DigiColor and ROIStore boards as devices and specifically indicated in this file that it was valid for the kernel to map in this space - it just happened to be in VME space. The addition to the *mem* file shown in figure 4 allows the boards to be *mmapped* with the code in figure 5.

```
/* entry allowing VME space to be mapped in
   specifically for the DigiColor and ROIStore boards */
#if IP5
        { VME_A24SSIZE, PHYS_TO_K1(VME_A24SBASE), },
        { VME_A32NPSIZE, PHYS_TO_K1(VME_A32NPBASE), },
#endif
```

Figure 4: Code added to kernel to allow it to map in VME space

```
if ((int) (roi_mem_ptr = mmap((caddr_t) 0, 0x80000, PROT_READ|PROT_WRITE,
           MAP_SHARED, fd, 0xc80000+PHYS_TO_K1(VME_A24SBASE))) == -1) {
    perror("unable to perform mmap for ROIStore on /dev/mmem");
    exit(1);
}
if ((int) (dc_mem_ptr = mmap((caddr_t) 0, 0x80000, PROT_READ|PROT_WRITE,
           MAP_SHARED, fd, 0xe00000+0x40000+PHYS_TO_K1(VME_A24SBASE))) == -1) {
    perror("unable to perform mmap for DigiColor on /dev/mmem");
    exit(1);
}
```

Figure 5: Code establishing pointers to VME space for direct access of boards

2.3 Digitization and Access

As can be seen in figure 3 we used a color camera that was connected to a DigiColor board capable of digitizing an image into several different signals. The DigiColor board can produce a composite signal, a monotone signal, a RGB signal, and others. Our goal was to choose a background, objects, and signal that allowed for easy object detection; we were not attempting to solve a vision recognition problem. We did however, want to design a realistic problem, and decided not to place a high intensity point source of light on the objects. Instead, by choosing an appropriate background we could use a threshold to determine if a particular pixel was part of an object. As mentioned the sheep are constructed using off-the-shelf Lego pieces. While we had eventually planned to design a cover or "wool" for the sheep, we wanted to be able to also detect the standard Lego objects. For each of the possible output signals the DigiColor can produce, we took a histogram of the pixel intensities. We wanted to determine empirically the signal that produced the sharpest and furthest spread peaks for the background versus the object. The best results were obtained by using the green digitized signal produced by the DigiColor in RGB mode and treating it as a monochrome signal. Although a slightly sharper distinction could be made by adding the red and blue signals, this would have required additional cycles in transferring the digitized image between the DigiColor and ROIStore boards.

We determined that the thresholding can work under significantly different lighting conditions assuming that the program is appropriately initialized. Initialization is accomplished by obtaining a histogram of the pixel intensity values for a particular lighting condition and choosing a threshold value. The first threshold value we tried was the midway between the peaks of the background and objects. While this provided a reasonable value, through experimentation we discovered a value eighty to ninety percent nearer to the edge of the object peak provided for less noise around the perimeter of the objects.

There are several different methods of accessing the memory in the ROIStore and DigiColor

boards. The Maxware primitives provide functions for accessing pixels individually or in blocks. Also by using pointers initialized as in figure 5 we could circumvent several layers of maxware code and access the memory in the ROIStore directly over the VME bus. This technique yielded a factor of three increase in access time when accessing individual pixels. We further determined that in order for the block accessing maxware primitives to outperform our direct access method, it was necessary to access on the order of 10000 contiguous pixels. These pointers could also be used to access an image plane in the DigiColor memory that could be used to create overlays for images. This allowed us to display computer representations of the objects directly on the monitor displaying the actual image, and was useful both in debugging and observing the program.

3 Decomposition and Real-Time Control

In designing any large program, it is important to consider the software engineering aspects as well as the actual programming. An additional problem in implementing a real-world application is ensuring the individual modules are programmed so they run quick enough and are scheduled within the required time window. In this section we describe the module decomposition and the user-level scheduler we designed using IRIX real-time primitives to yield the desired behavior.

There were many hardware components used to implement the shepherding application. Figure 3 illustrates the connections between them. We used a puma robot arm to manipulate the sheep on the table. Control of the arm requires a software package called RCCL [3] that runs on a Sparc workstation. While we could have connected the vision processing boards to the VME bus on the same workstation and run all the vision processing, planning, display, and other functions on that same workstation, there was considerable motivation to implement all but the RCCL control on a more powerful SGI multiprocessor.

The decision to use the SGI was based in part on the realization that the vision processing portion required both intensive computation and extensive access to the ROIStore board on the VME bus. The SGI Challenge series with R4400 chips clocked at 100 MHZ and a 1.2 GByte bus, provided both faster processors and a faster bus. In addition to the increased power of the SGI, as mentioned in the introduction, we were interested in using the shepherding application to study and verify the issues involved in designing parallel real-world applications. Therefore, the vision processing boards were placed on the SGI. The goal was to place all the code possible on the SGI requiring as little as possible of the Sparcstation. All the vision processing code, the planning code (including where to send the robot arm), and the display code, was implemented on the SGI. Since the RCCL software needed to run on the Sun workstation, a method of communication between the Sun and SGI was required. The SGI performed the majority of the computation and simply sent desired robot arm coordinates to the Sun. This placed as little of a burden on the Sun as possible. All the Sun workstation needed to do, was to perform a transformation between the image coordinates sent by the SGI and the robot world space, move the arm, and send back confirmation of success or failure.

The task running on the Sun workstation was straightforward: an infinite loop was setup to wait for a command from the SGI, perform a robot move, and send confirmation. On the SGI however there were may software modules that needed to be meshed. The module that coordinates and schedules the processes on the SGI is our user-level scheduler. It interacts with Ephor in order to facilitate user implementation. The user-level scheduler was designed in order to try new techniques and interactions between Ephor and the scheduler both rapidly and without kernel modification. After describing our user-level scheduler and the salient aspects of Ephor (a more detailed description of Ephor can be found in [4, 5]), we provide a description of the different modules comprising the shepherding application. The vision and manipulation modules receive more detailed treatment in later sections and thus are only briefly described here. Since the shepherding application ran on a

```
#define BEGIN_PROC(GOAL,ID,STRID) \
  int ftime1,ftime2;\
  int *cpu_times, *cpu_index;\
  cpu_times = goal_list[GOAL].technique[0].cpu_times;\
  cpu_index = &(goal_list[GOAL].technique[0].cpu_index);\
  while(start_signal == 0);  /* delay so proc id stabilizes */\
  if (verbose) printf("%s id %d\n",STRID,ID);\
  if (sysmp(MP_MUSTRUN, goal_procs[GOAL].proc) < 0)\
        printf("error: failed to assign processor %d to %d i am %s\n",\
        goal_procs[GOAL].proc, GOAL, STRID);\
  goal_procs[GOAL].migrate = 0;\
  setblockproccnt(ID,0);\
  schedctl(NDPRI,0,NDPHIMAX);\
  while (1) {\
    blockproc(ID);\
    if (goal_procs[GOAL].pri_add)\
      schedctl(NDPRI,0,NDPHIMAX+goal_procs[GOAL].pri_add);\
    if (goal_procs[GOAL].migrate) {\
      if (sysmp(MP_MUSTRUN, goal_procs[GOAL].proc) < 0)\
        printf("error: failed to assign processor %d to %d\n",\
        goal_procs[GOAL].proc, GOAL);\
      goal_procs[GOAL].migrate = 0;\
    }\
    ftime1 = fine_clock;
```

Figure 6: C code for the BEGIN_Proc macro

multiprocessor, each module was given an individual processor. While highly cpu intensive modules (the vision processing and planning modules) could be parallelized as in our simulator, we found the real-world bottleneck was in the robot and communication, thus each module was assigned only one processor.

3.1 User-Level Scheduler

The user-level scheduler allowed Ephor to control the placement and timing of the tasks. In turn Ephor provides the user with a clean interface allowing easy specification of when and how frequently to run a particular task. In addition Ephor interacts with the user-level scheduler providing dynamic task selection, parallel process control, and more mechanisms for the SPARTA programmer [5].

It is necessary to have cooperation between the tasks and the user-level scheduler in order to simulate a real scheduler. The code for this functionality is placed in a header defined by a macro. Providing this macro removes responsibility from the user for providing the cooperation between Ephor and the user-level scheduler. All each task needs to do, is to place a BEGIN_PROC (see figure 6) statement at its beginning. BEGIN_PROC is a macro allowing the user-level scheduler to place this task on a specific processor at a given time. Combined with the END_PROC (see figure 7) it allows for very precise timing.

The BEGIN_PROC macro sets each task into an infinite loop with a *blockproc* statement at the beginning followed by the code for the work. *Blockproc* causes a particular process to block until an *unblockproc* command is issued. The processes should be thought of as light-weight threads as

```
#define END_PROC(GOAL) \
   ftime2 = fine_clock;\
    (*cpu_index)++;\
    cpu_times[(*cpu_index)%3] = ftime2-ftime1;\
    }
```

Figure 7: C code for the END_PROC macro

```
goal_list[VISION_PROC].periodic = TRUE;  /* indicate this task is periodic */
goal_list[VISION_PROC].rate = 1; /* run every INTERVAL */
```

Figure 8: Code for establishing a periodic task

they share address space and differ only in necessities such as the program counter, stack space, etc. After initialization a set of processes are created, one for each of the tasks. They have each executed the *blockproc* command appearing in the macro header. When the user-level scheduler needs to run a particular task it unblockprocs the (light-weight) process associated with that task.

Additional code in the BEGIN_PROC macro allows the user-level scheduler to also have control over the processor the task runs on. Each task (via code in the macro) executes a *sysmp(MUSTRUN, proc[my_id])* command. The task executing this commands runs on the *proc* processor specified by the user-level scheduler. The array *proc* is set by the user-level scheduler for each task and can be changed dynamically. Another concern was getting accurate timings for the different tasks for scheduling purposes. The clock provided by the SGI only gave ten millisecond granularity for standard processes. While millisecond granularity was available to higher priority tasks, they are non-preemptable and we wanted the ability to allow the user-level scheduler to block tasks. An extremely high granularity clock was established on a distinct processor by using a shared variable and continually incrementing it. This provided 121 nanosecond resolution and was accurate to under one percent. All tasks were timed by checking this value in the BEGIN_PROC macro of the task and rechecking it in the END_PROC macro.

There are still more subtleties in ensuring the above mechanisms behave as expected. To guarantee the desired behavior, some additional IRIX real-time primitives were used. The processors were restricted (*sysmp(MP_RESTRICT,i)*) to running only the processes that had been assigned to them by the user-level scheduler. To ensure the tasks ran in the correct order the priorities associated with the processes were modified appropriately by a method similar to determining the processor to run on. The scheduler also had to track the execution times of the tasks and ensure that it placed only those that could run in the allocated time period.

The important aspect of Ephor and the user-level scheduler from the application programmer's perspective is the ease with which they can specify timing constraints and priority concerns, and the increased performance achieved by Ephor's automatic mechanisms. The user fills in a shared data structure indicating whether that task is to be run periodically and it's rate (e.g. for the object finding task) or whether it will be run in response to an environmental stimulus (the manipulation task). The lines of code appearing in figure 8 shows how to indicate a task is periodic while the code in figure 9 shows how to initialize an environmentally responsive task and later how to indicate to Ephor that that task is to be run.

```
goal_list[SAVE_SHEEP_GOAL].periodic = FALSE;  /* indicate task is to be run
                                                 when application
                                                 specifies */
/* elsewhere in the program the user indicates this goal is to be solved by
   executing the following line of code */
to_do[index_of_goal_to_be_solve] = 1;
```

Figure 9: Code for establishing an environmentally responsive task

3.2 Display Process

The *display* process was created primarily to provide a nice user interface allowing the programmer to view the computer's representation of the real world. This allowed both easier debugging and easier development. The interface also allows the user to control some of the actions of the application, e.g., whether to track objects to find them or scan the whole image to find them.

The *display* process starts by opening an X window with a portion of the window for displaying sheep positions and another portion containing action buttons. When blobs appear in the real-world scene, the *display* process obtains their image positions and displays them in the window as a circle. The area of the circle matches the area of the blob as found by the vision processing task. As the blobs move, their locations in the X window also move.

The X window can be used to control the computer program. One of the buttons appearing in the X window is a scan button. This effectively pauses the vision processing algorithm, i.e., it no longer searches for blobs in the image. This allows the user to enter the image and move objects around. Also as part of this mode, the user can move a computer generated box around the monitor displaying the image. This is useful in performing the initial robot calibration, where it is necessary to correlate points in the image to points in the robot's space. Other buttons include a pair indicating whether the program is currently tracking images and performing a perimeter search (trackify option) or whether the computer is trying to find blobs in the entire image (blobify option). Other options include whether to perform a full scan of a blob or just an axis scan. There is also a debug button allowing information (e.g., variables) to be dumped to the screen. This can be used to actually find a bug in the program or just a quick method to print out transient information about the objects. The details of these options and tools, and the tradeoffs and usefulness of some of them will be discussed in later sections.

3.3 Communication Process

There is communication between the SGI program and the robot control program on the Sparcstation. Communication is set up with sockets allowing transmission of short messages containing target image points and the verification of a robot move. A separate process is used since socket communication is a bottleneck in the system. If the *communication* process were joined with the *planner* process, a lot of planning time would be lost while waiting for message replys. By spawning off a separate *communication* process, the system is allowed to continue processing visual input and planning intercepts while waiting for communication to complete.

When the *communication* process is first started, it has the robot control program (running on the Sparcstation) perform all necessary initialization. After initialization, the *communication* process spins in a loop waiting until the planner has decided on a robot manipulation. When a target position and manipulator orientation for the robot has been planned, the *communication* process sends the target and orientation to the Sparcstation for processing, and waits for a reply

verifying the arm movement. Upon confirmation, the *communication* process notifies the *planner* process, via a flag, that the motion is complete. It then waits for the next robot position to send.

3.4 Planner Process

Determining the next sheep to save can take up a considerable amount of time depending on the strategy employed. It is possible to imagine a broad spectrum of possibilities ranging from picking the first sheep found moving away from the center, to exploring all possibilities of the next n sheep to save (exponential). One of the mechanisms in Ephor allows a user to program many different possibilities and will automatically and dynamically select the most appropriate one at runtime [4]. Unfortunately with the hardware in our laboratory this decision was relatively uninteresting because we could not physically get that many sheep on the table. (The puma arm we have is both slow and does not have much reach). However, this did bring up other tradeoffs based on the deficits of the robot arm, e.g., it had problems reaching one corner of the field so it was best to herd sheep away from it earlier than for the other corners. Another difficulty in the laboratory was the fact that the robot was extremely slow in comparison to the processors, allowing considerable computation for each manipulation move. We still did investigate the tradeoffs between the planner described above and one that exhaustively searched the entire space.

The real-world shepherder has two versions of the planner: the single sheep version and the multi-sheep version. Both are described in the following paragraphs.

The single sheep planner is designed to restrict the motion of a single sheep. When the sheep is spotted, an intercept point is found on the field boundary, the robot positions itself at the intercept point in the elevation plane (a plane far enough above the field to allow free movement of the robot arm) with the manipulator oriented to catch it, and then the manipulator is lowered to the object plane (the plane of the field). Until the sheep reaches the intercept point, the position and orientation of the manipulator are corrected to compensate for slight changes. When the sheep reaches the manipulator, it is reoriented toward the center of the field. Finally, the manipulator is removed from the path of the sheep and placed in a location such that the arm will not obscure any of the field (it is placed in the lower right corner). This *planner* process is currently very simple but is sufficient to contain one sheep. It allowed us to verify that the other hardware and software components functioned as required.

To contain more sheep, a more complicated planner is required. First, a sheep that needs to be saved is chosen according to the following criteria: 1) it will leave the scene sooner than any other sheep, and 2) it is not already headed toward the center of the scene. If no sheep fits the conditions (no sheep need to be saved yet), then the arm moves to the lower right corner of the scene to prevent sheep from being obscured.

Second, an intercept is calculated according to arm position and velocity, sheep position and velocity, and a delay long enough to allow the arm to move from the elevation plane into the object plane. If the intercept is outside the field, or if it intercepts the expected location of a second sheep, then the first sheep is ignored and a new sheep is considered for rescue.

Third, after the intercept has been accepted, the arm is moved to that location in the elevation plane. Once the move has been made, the intercept is rechecked to ensure no sheep are directly under the arm (it is possible that before the arm moves no sheep other than the target will be at the intercept, but during the move two sheep could collide directing a sheep to the intercept point). After the second intercept verification has been made, the arm is lowered into the object plane (directly over the target sheep), the sheep is redirected toward the center of the scene, and the arm moves into the elevation plane.

Finally, this process starts all over again by either going to save another sheep, or moving to the corner of the scene if no sheep currently needs to be saved.

3.5 The Vision Process

The *vision* process performs a search on the image data for groups or blobs of pixels indicating the positions of the sheep; this is known as the blobify routine. The *vision* process is run at a specified regular interval and controlled by the *scheduler* process. The *vision* process is run 20 times a second, so the blobify routine must be fast enough to complete once during every interval. As described later, we needed a high scan frequency to meet the constraints placed on the real-time vision processing portion by the shepherding application.

3.6 The Manipulation Process

The *manipulation* process resides on the Sparcstation controlling the robot arm. It is sent an arm position and manipulator orientation specified in image coordinates. First, the *manipulation* process converts the image coordinates into robot world coordinates (this calculation is described later). Second, it makes sure the desired speed is possible. This is not straightforward since speed is specified in Cartesian coordinates (pixels per second and radians per second), while the arm speed is constrained by the six independent joint velocities. The check is done by performing inverse kinematics to calculate new joint positions. Then each joint is checked to make sure none exceed their maximum velocities. If a joint velocity will be out of bounds, then the overall speed is reduced to allow the arm to perform the movement. Once the arm speed is verified, the move is made and the *manipulation* process sends a message back to the SGI verifying the new position of the arm.

4 Vision

Detecting and tracking moving objects in the real world places a set of timing constraints on the vision processing portion of an application that differ from standard vision processing. Additionally, the shepherding application requires very accurate velocity. To satisfy the real-time constraints, the vision processing algorithm must execute both quickly and be predictable from execution to execution to allow the scheduler to calculate a suitable interval. For example, if the processing time spent on finding objects varied significantly, then that portion of the algorithm may not finish in the expected interval causing stale data to be used and thus invalidating the velocity prediction portion of the algorithm. The calculated velocity is used in the shepherding application to predict where the object will be many (perhaps one hundred) steps in the future. Thus, in addition to tracking the current position of the objects, it is very important in the shepherding application that an accurate velocity be found as any error will be multiplied many times as the future position is predicted. To obtain an accurate velocity we need to maintain a high resolution image, i.e., subsampling yields less accurate positions and thus less accurate velocities. In the shepherding application there are many objects (sheep) that could be in the field simultaneously. It was necessary to be able to track multiple objects and be able to associate blobs in the present image to objects from the previous images. It was also possible that some of the objects would be obscured for variable lengths of time. It was therefore necessary to develop an algorithm capable of handling obscured objects. The vision portion of the shepherding application needed to be able to track multiple objects rapidly, run with small time variation, and be able to produce very accurate velocity prediction.

We present an overview of the vision algorithm here and describe each phase in more detail in the upcoming sections. The first stage in the algorithm is to determine where the blobs are. Blobs represent probable objects. Once these have been obtained it is necessary to associate the blobs with objects. In many cases this is a fairly simple operation since using a high scan rate prevents the objects from shifting significantly between snapshots. However, it is possible that due to object collisions or an object being obscured this initial match is not successful. To handle these situations,

obscured objects are given projected positions and "tracked" while "new" objects are remapped onto the old expected objects. After a complete pairing has been accomplished the object positions are used to calculated a preliminary velocity. This velocity however is noisy due to the very noisy movements of the sheep and the noise associated with the camera. The velocities are passed through a double $\alpha-\beta$ filter with distinct parameters and different sampling rates for each filter. The output of the second filter is taken to be the "true" velocity and the data structure associated with that object is updated. This velocity can then be used to predict future positions of that object.

4.1 Blobify

The first stage in the vision algorithm is to group the pixels above the threshold value into blobs. This blobification process is simplified because we used a solid background with a pixel value lower than that of the objects. The actual objects (sheep) we used are displayed in figures 10 and 11. As mentioned earlier, the object finding algorithm worked on the uncovered sheep. There were, however, two strong motivating factors for providing "wool" for the sheep. The bare Lego sheep provided for non-elastic collisions that tent to clump sheep throughout the field making for an uninteresting shepherding problem. The jagged edges and wires produced an additional source of noise (the number of detectable pixels had a much higher variance). Further, the appearance of bare Lego sheep is extremely susceptible to variations in lighting conditions due to the shiny surfaces of the pieces. The wooled sheep provided a more consistent size under similar lighting conditions in different portions of the image and were also more consistent across different lighting conditions. Our motivation in placing the wool on the sheep was to allow us (when implementing our real-world application) to focus on the real-time tracking and real-time association problems rather than the object detection problem. Should the need occur to track natural, the simple thresholding function would be replaced with a suitable object detection function.

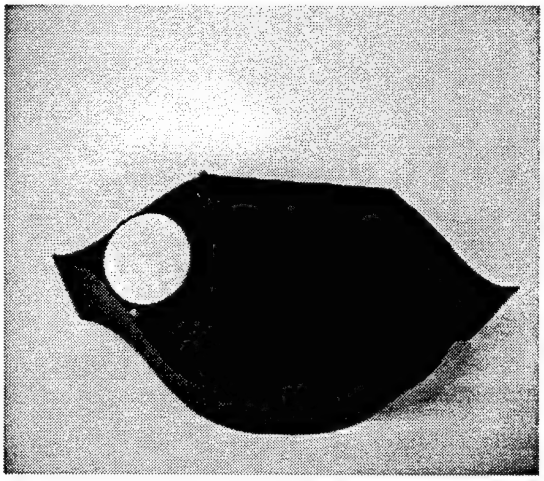

Figure 10: A Lego sheep covered.

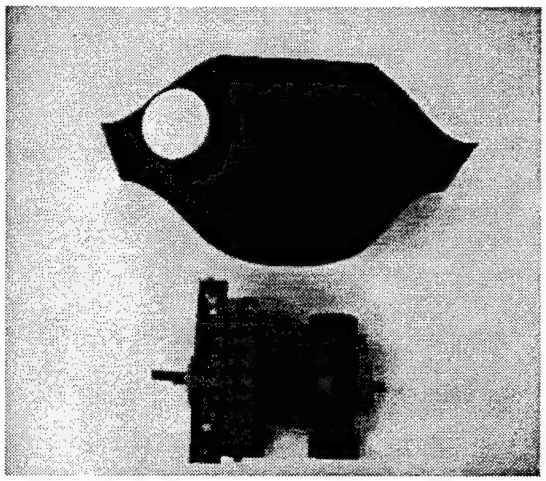

Figure 11: A Lego sheep cover next to a sheep

Blobs are searched for in a 512×482 image stored in the ROIStore memory as a one-dimensional array after being digitized by the DigiColor. Blobifying the image can easily be the most time consuming stage since each pixel must be referenced over the VME bus. It was at this stage that we had to utilize intelligent algorithms that differed from standard vision processing ones. We needed to blobify the entire image quickly and at high resolution. Our goal was to be able to process the entire image between twenty and thirty hertz. To reference every pixel in the ROIStore required .8 sec. Even performing a bcopy from VME space to main memory of the image required .3 sec,

plus the time to access them from memory. Clearly it is not possible to examine every, or even a significant percentage of, the pixels and meet a twenty hertz constraint. We therefore developed a two phase examination of the pixels.

Conceptually the algorithm is broken into two phases. In the first phase we perform a very sparse subsampled search. We use information about the size of the blobs and try to make the search in this phase as sparse as possible. The object of this phase was to produce plausible locations for objects. A sparse two-dimensional shadow array (a sample one is shown on the left side of figure 12) of the actual image is kept in main memory and upon identification of a possible object a cell is marked. A true value in the shadow array indicates a possible object for expansion in the second phase(the X's in figure 12). The X's in figure 12 indicate cells in the sparse array that would contain a pixel above threshold. Notice that an X occurs in the sparse array only if there is a corresponding pixel in the image that would be in that cell.

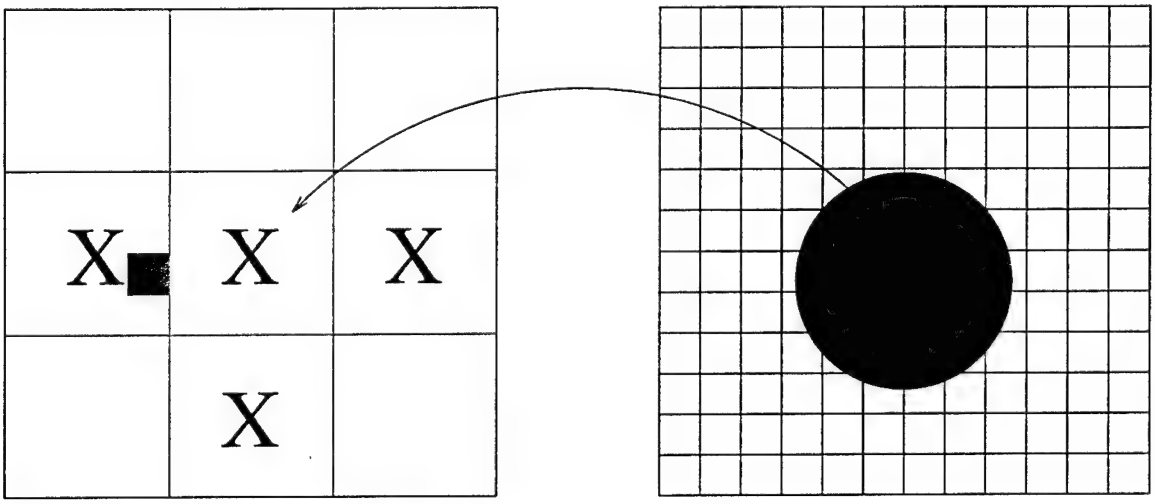

Figure 12: The actual image (on the right) and its shadow array (on the left)

The second phase involves performing a high resolution search on the pixels in the actual image corresponding to those marked in the shadow array in the first phase. The goal is to determine the centroid of the object in both the x and y axes. This will be used as its position to calculate velocity and then will be passed through a double $\alpha - \beta$ filter. There are several possible methods to perform a search. One algorithm is based on the simplifying assumption that the objects are close to circular. The first step in this algorithm is to scan up and down from the original point. The algorithm performs a scan in each direction until it determines, by pixel intensity value, it is at the end of the object. This obtains a first preliminary vertical line as shown in circle **1** of figure 13. A center point is determined for the vertical line and a horizontal scan is performed in each direction, again until the end of the object is found. This operation finds the horizontal diameter of the object (see circle **2** of figure 13). As a check that the object was shaped close to expected (a circle) a third vertical scan can be performed as in circle **3**. If the ends of the object are approximately the same distance away from the horizontal diameter, then the center of mass is the point at which the two diameters cross. It is possible that in performing the check the algorithm determines the object was not circular.

In many cases, as was true for our shepherding, the circle assumption could not be made. Even though the spots on the sheep's wool were circular (figure 10), they could become obscured. While becoming obscured, the spot loses its circular shape. We performed experiments and determined that even with reasonably slow moving sheep and a slightly faster robot arm, this effect could throw

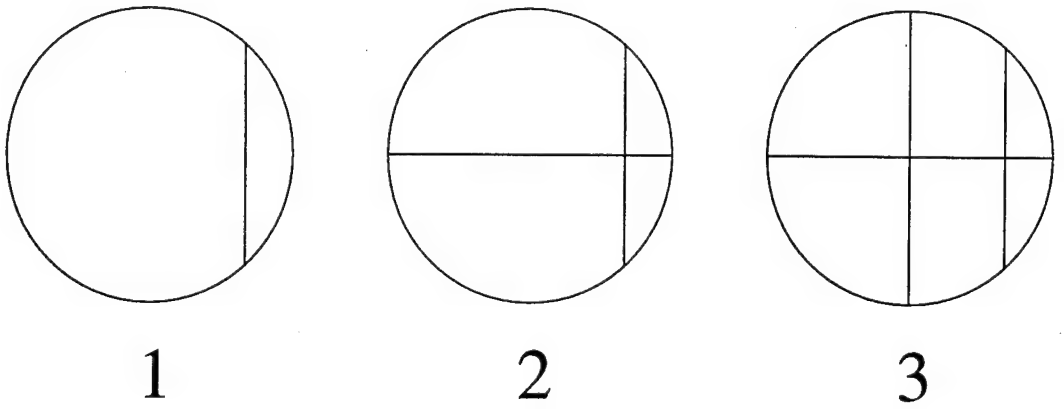

Figure 13: The lines created by the quick circle method

the velocity off by up to a factor of five. This is because the center of mass changes much more rapidly as an object is becoming obscured. It may well also be the case the image being blobified does not consist of circular objects. To handle non-circular objects, a recursive search of all the pixels in the object is performed about the point corresponding to the one in the sparse array. Since speed was of primary concern, rather than using function recursion, we implemented a stack of pixels and performed the entire recursive search with one function call. A step is made in each of four directions until the end of the object is encountered. A cumulative total of x and y values and number of points was kept. After the entire object has been explored the x and y sum is divided by the number of points to determine the x and y centroids.

Sampling rate	blobify	trackify
4 pixels	26 ms	18 ms
8 pixels	17 ms	8 ms
10 pixels	12 ms	4 ms
12 pixels	9 ms	3 ms

Figure 14: Comparing blobify to trackify

In reality, as an optimization, the phases are combined. As soon as a pixel is found in the sparse scan, we pause phase one and perform a high resolution scan about this point. After the object is explored in the high resolution image the corresponding cells are marked as invalid (do not expand) in the sparse shadow matrix as in figure 12. This eliminates having to check to make sure duplicate objects are not produced. It also eliminates extra references to pixels in the ROIStore, i.e., if the stages were not combined there would be many more forward examinations of each possible hit in the sparse array. The result is a high resolution blobify routine that runs very quickly due to a reduced search space and a fast interactive version of a detailed, recursive, search function. When the blobification process completes, an array of blob positions denoted by x and y centroids has been updated.

Instead of searching for blobs in the entire image, an optimization is possible if we assume objects can only enter the field from the perimeter (sheep can't fly). We will call this algorithm *trackify* since it involves looking for object based on where they were in the last image. The center of each object is used as the initial point in the sparse matrix and a high resolution scan is performed using it as the origin. Then a general scan, as previously described, is performed around the perimeter.

For this to be valid the objects can not move more than the distance equal to their radius between frames, otherwise the initial point (the center of the object from the last image) will not be part of the object. This is a reasonable assumption. In our case sheep moved two or three pixels a second and they were about sixteen pixels in diameter. We scanned at 20Hz. Thus, we had about 80 images before the center of the object would no longer be a point anywhere in the object.

Driving the hardware a this rate is challenging. The DigiColor can digitize half (every other line) the image at 60Hz. Thus a complete new full image image is available at 30Hz. Trying to synchronize the algorithm with the scan rate (by checking appropriate flags) and the rest of the code may not have been doable at 20Hz. Instead, we perform a continuous scan disregarding where the DigiColor is in writing the image. This could produce a situation where garbled data was being used if we happened to be reading in the area the DigiColor was writing. To avoid this difficulty we scan backwards in ROIStore memory while the DigiColor writes the digitized image forwards. In this way, the possibility of accessing pixels being written is minimized and the area of potential overlap is reduced to a few pixels.

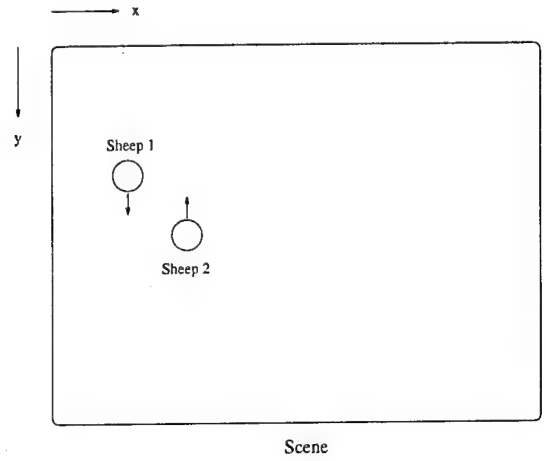

Figure 15: A sample scene with two sheep and their velocity vectors.

The difference between performing a trackify and blobify operation can be significant. As illustrated in figure 14, there can be a significant difference between performing these two operations. The sampling rate indicates the ratio between the sparse matrix and the actual image. For example, a sampling rate of eight indicates every eighth pixel was examined in the first phase of the algorithm. The numbers in the table were gathered performing a high resolution recursive search with six objects in the image each with of diameter of approximately sixteen pixels. Performing the full high resolution scan versus making the circular assumption adds only about .3msecs per blob or 1.8msecs for the numbers in the table. Since this number is small we always perform the full search when blobifying in the shepherding application.

4.2 Associating Blobs to Objects

The next step is to associate blobs with objects (sheep). Blobs are associated with objects for several reasons. The planner requires velocity estimates as well as position estimates. A blob is only a snapshot in one image. Storing consecutive positions in an object structure allows continuous velocity estimates to be obtained by using a filter. This is important because at any time the planning process may need to know the position and velocity of a particular sheep. There are, however, difficulties that arise with attempting to provide continuous positions to the filter. When

the robot arm moves over a sheep the sheep becomes obscured and a blob is no longer reported for that object. If the blob was being stored as an object we can continue to estimate where that object will be. Objects also admit the ability to disambiguate two blobs with velocity vectors as shown in figure 15. As sheep 1 moves down past sheep 2 the blobifier will form the blobs in a different order. This figure illustrates that it is not possible to simply feed the same blob center positions to a filter or any other permanent data structure; it is important that an intelligent mapping between blobs and objects occur. In order to update sheep positions while obscured, calculate velocities, and associate a sheep with a single, nonchanging set of values, blobs need to be associated with specific objects.

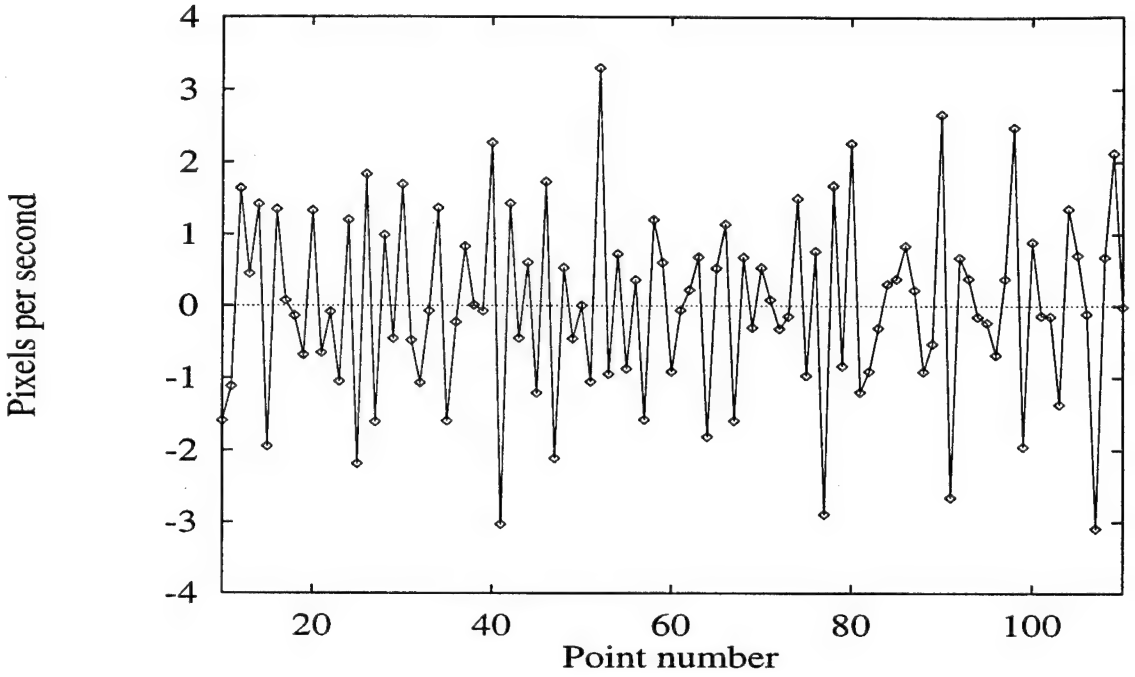

Figure 16: Velocity for a stationary sheep

The task of associating blobs to objects is non-trivial. The process is broken down into four stages executed in the following order: 1) associate visible blobs to known objects, 2) map unassociated blobs to recently unobscured objects, 3) estimate the positions of obscured objects, and 4) map unassociated blobs to new objects.

1) Associate Visible Blobs to Known Objects

This step is straightforward. Since we use a high scan rate, objects move less than a pixel from snapshot to snapshot. A pointer is kept to the blob a particular object was associated with last time. If the blob's new position is consistent with the last position and velocity estimate of the object, then the association is kept. If there is no blob whose position is consistent with the object, then it is assumed that the sheep has become obscured and the object is marked as such. Also, if the object's new position is outside of the field (the sheep escaped the confined area), then the object is marked as dead and removed from the list of objects.

2) Map Unassociated Blobs to Recently Unobscured Objects

Once blobs have been associated with objects, there might be some unassociated blobs left over. This can occur for one of two reasons: either a blob has recently been obscured and has just move out from under the robot arm, or an entirely new sheep has appeared on the table. A search is made through all the objects that have been marked as obscured to determine if one of those might match the blob in question. This match is determined by using the estimated positions and velocities of obscured objects. If the blobs current position is within a predetermined percent of where the object was expected to be, an association is established. A positive association may not be made if while the blob was obscured it was manipulated back toward the center, or if it collided with another sheep. The allowed percent is increased (effectively widening the search) until a positive match is found. Our double pass filters allow for the velocity estimate to quickly track the object's new heading, so even if it is the case that the object has been turned around, the velocity estimate will soon (within two to three snapshots, or about a tenth of a second) correctly reflect the object's true velocity.

3) Estimate Positions of Obscured Objects

When a sheep becomes obscured, it is necessary to estimate its position according to its last measured position and velocity. This is done simply by multiplying the amount of time the object has been obscured with it's last estimated velocity. This is another reason why it is important to have very accurate and quickly determinable velocities. The position estimate kept when obscured is used when trying to reassociate a blob detected on the field. It is also used by the planner to determine the next sheep to save. It is possible that an obscured sheep needs to be reoriented so that it won't escape the field. If the estimated position of an object is ever outside of the field, it is marked as dead and the sheep is removed from the object list.

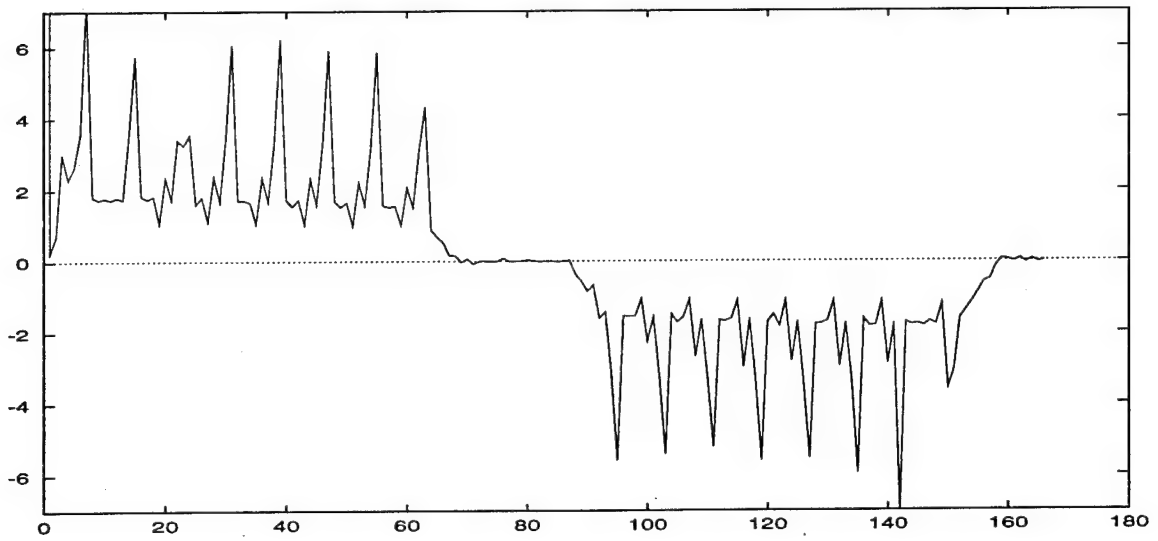

Figure 17: The actual measured velocity of the toy sheep where the x-axis is in twentieths of a second and the y-axis is pixels per second.

Plant noise filter 1	0.01
Measurement noise filter 1	0.1
Plant noise filter 2	0.05
Measurement noise filter 2	0.01

Table 1: The parameters used in both of the $\alpha - \beta$ filters.

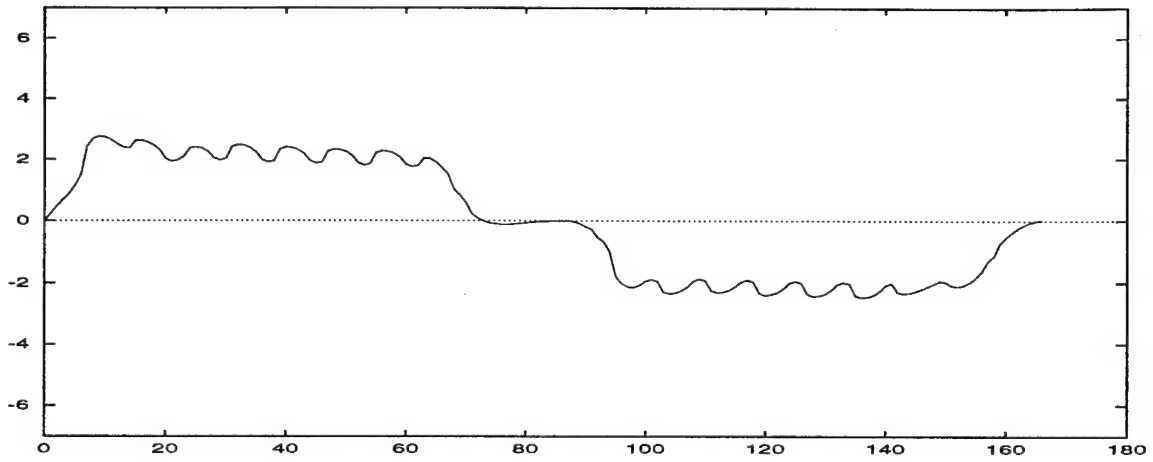

Figure 18: The output of the first $\alpha - \beta$ filter.

4) Map Unassociated Blobs to New Objects

By this last stage in object to blob association, if there are any unassociated blobs, they are assumed to be new and are mapped to a new object. Hence, a new object is added to the list, the position information is added, and the filtering process begins. In our shepherding application, the "trackify" assumption is that objects are placed only on the perimeter. The "blobify" algorithm allows new objects to appear anywhere in the scene. For instance, a sheep entering the field may slide under the robot arm and be obscured until it has already crossed a portion of the field. While careful placement would avoid this difficulty, we designed the blobify algorithm to handle such a scenario.

4.3 Filtering

Once blobs have been associated with objects, another operation is needed before a final velocity is ready for the planner. It is necessary to filter the data. Figure 16 shows velocity data for a sheep standing perfectly still. With no noise, the velocity would be a stable zero. Since the object is stationary, this figure represents the sensor noise introduced by the camera and digitizer. The camera noise represents a significant portion of the total noise. The Lego sheep introduce plant noise since they do not move at constant speed; the exact quantity however is difficult to capture. An examination of figure 17 (this figure shows the velocity of a sheep moving in one direction, a 180 degree rotation, followed by velocity in the opposite direction) shows periodic noise patterns indicating that the sheep are adding their own noise on top of the measurement noise produced by the camera. Successive figures in this section represent combined plant and sensor noise.

To reduce the effects of plant and measurement noise, an $\alpha - \beta$ filter was used to incorporate new data into old [2]. The sheep occasionally change direction, even by as much as 180 degrees (when

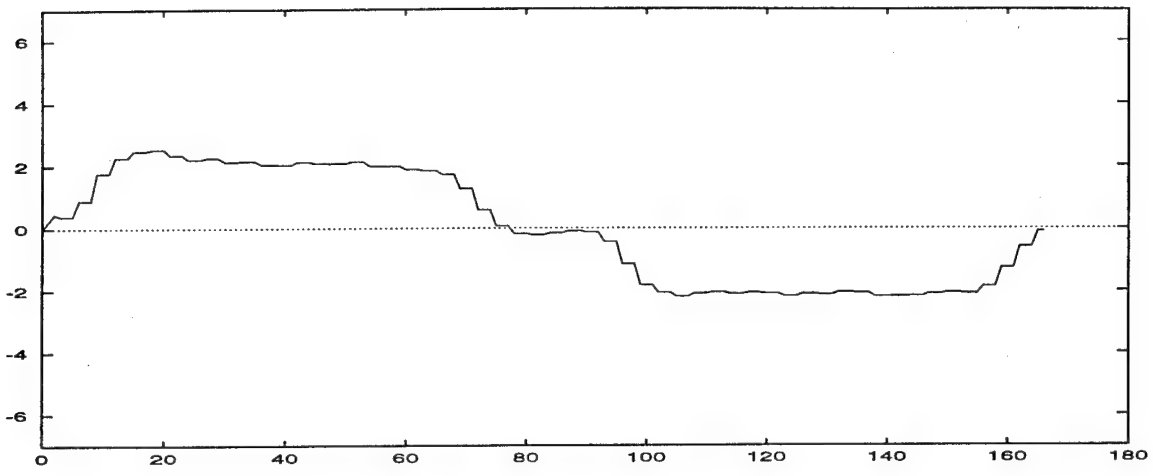

Figure 19: The output of the second $\alpha - \beta$ filter.

being manipulated back towards the center), or less when deflecting off other objects. As stated, the planner needed very consistent velocity estimates. The requirements of the filtered velocity were that it be smooth and representative of the averaged true velocity and respond quickly to changes in the object's motion. An $\alpha - \beta$ filter has two parameters that can be set. These represent how much confidence should be extended to the data. If the filter is set to have a high degree of confidence in the data, the filtered output will very closely resemble the input data, and any change in direction will quickly be reflected in the filtered output. However, if the input data is very noisy then the filtered output will still be quite noisy. If the parameters are set reflecting little confidence in the data, then the filtered output is smooth but responds slowly to changes in direction. We were thus faced with a dilemma when choosing the parameters. In fact, several experiments confirmed our suspicion that there would be no appropriate choice of parameters for a single $\alpha - \beta$ filter.

To solve this problem, we ran the noisy input data through a double $\alpha - \beta$ filter with different parameters for each filter. The first filter's parameters (see table 1) are set assuming small plant noise. This is so the filtered output will quickly respond to changes in directions of the objects. The original data appears in figure 17. The output from the first filter appears in figure 18 and although it is still fairly noisy, it closely tracks changes in direction of the sheep. As mentioned, the eventual output needed to be smooth for accurate prediction by the planner. To achieve this, the output of the first filter was used as input to a second filter. The second filter assumed a higher plant noise (for greater smoothing) and a much lower measurement noise (the data had already been filtered so it shouldn't be as noisy as the original data). Additionally, we subsampled the first filter data, taking every third point, to produce an even smoother curve. The output of the second filter appears in figure 19. The output of the second filter fulfills the requirements: it is quite smooth and responds quickly to changes in direction of the sheep.

To see how well the output of the second filter mimics the original input data, as well as the intermediate stage, a graph overlaying the data presented in figures 17, 18, and 19 is presented in figure 20. The velocity from the second filter fulfills the two requirements of having a small delay in adjusting velocities when there is a change in direction and of having a smooth profile.

To gauge the accuracy of the filtered velocity estimate, we ran several experiments in which a sheep was introduced into the field and the filter was allowed to establish a velocity estimate. A pseudo-planner asked for the sheep's velocity and predicted the sheep's position n seconds into the future. The pseudo-planner waited for the n seconds and asked for the position of the sheep. We recorded the difference of the predicted position of the sheep with the actual position. We performed

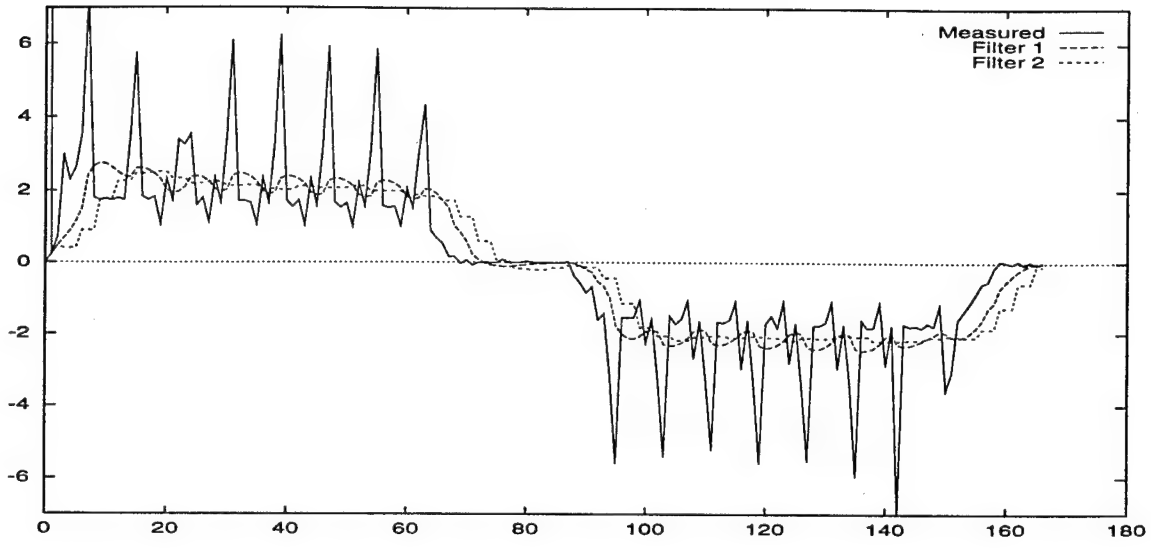

Figure 20: The measured, first filter, and second filter velocities.

this experiment with n taking on values of 1, 2, 4, 8, and 16. For each n, we ran six experiments and took the mean of the absolute values of the difference between the predicted and actual position. The graph in figure 21 represents the results. The y axis is in pixels. The approximate velocity of the sheep was .8 pixels per second. Remember, the diameter of the sheep is about 16 pixels. The graph indicates very accurate prediction and the accuracy is not significantly affected by increasing time. This is true for two reasons: the plant noise (the velocity) is periodic so when integrated over time is predictable, and there is a large amount of error arising from sensor noise (the measurement of the positions of the sheep). These experiments were run with a 20HZ scan rate; 16 seconds represents multiplying the estimated velocity by 320 to obtain the predicted position. These results clearly indicate the estimate velocity from the double filter is very accurate.

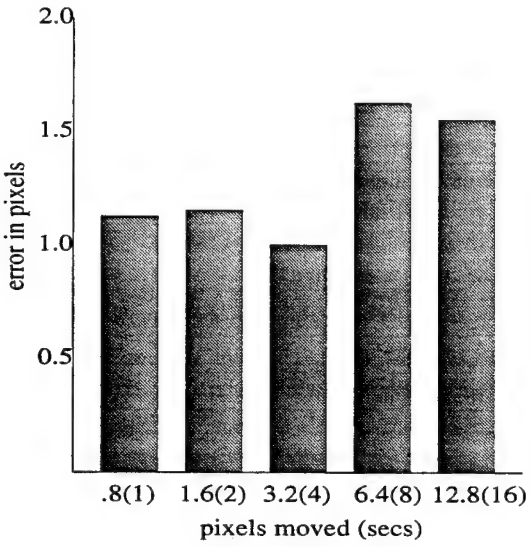

Figure 21: Error in predicting a sheep position.

5 Manipulation

One of the most difficult aspects of real-time manipulation is that the robot arm is extremely slow and unpredictable. The program that controls the Puma robotic arm runs on a Sparcstation using the RCCL robot control software. As we previously described, this program is sent requests from the SGI where the vision processing and planning occur. The task of the program running on the Sparcstation is to move the arm to a specified (x, y) point with a given orientation θ. It is not straightforward since the SGI specifies a target point in two-dimensional image co-ordinates. The manipulation program must use a transformation matrix to convert between the requested image coordinate specified by the GSI and the robot's three-dimensional world coordinate frame.

To simplify the problem, the robot is required to work in only two given z-planes: the object plane where it can manipulate the sheep, and the elevation plane where the arm can move freely without bumping the sheep. Due to the specification of z-planes, the transformation requires a 2×3 matrix.

To create the transformation matrix, three sample points must be taken to set up a correspondence between image and world points and mark the object plane, and a fourth point is taken to specify the elevation plane. Once the sample points have been taken and the transformation matrix is created, the operation in equation (1) must be made to convert image points into world points.

$$\begin{bmatrix} T_{11} & T_{12} & T_{13} \\ T_{21} & T_{22} & T_{23} \end{bmatrix} \begin{bmatrix} x_I \\ y_I \\ 1 \end{bmatrix} = \begin{bmatrix} x_W \\ y_W \end{bmatrix} \tag{1}$$

In the initialization phase, the elements of the 2×3 matrix shown in equation (1) must be found. It is clear that that can be done by getting three world points and their associated image points and solving the set of three equations to find the three unknowns (three unknowns per row of the transformation matrix). The object plane is specified by the user in the first image to world point correspondent.

It is also simple to find the world co-ordinate orientation of the gripper given the image co-ordinate orientation. A vector from the origin of the image can be easily computed given an orientation (in radians). Then, using the transformation matrix, the corresponding world vector can be found. With this information it is possible to orient the manipulator in the world.

The final aspect of robot control is the speed of the arm. Fortunately, RCCL provides a function to set the time it takes to get to a target point. Given the current location of the arm and the destination point (in image co-ordinates), we can use this function to find the travel time of the arm. Unfortunately though, this is sometimes in error and an allowance for this possibility was required. The planner also has access to the arm speed constant allowing it to estimate how long a requested move will take.

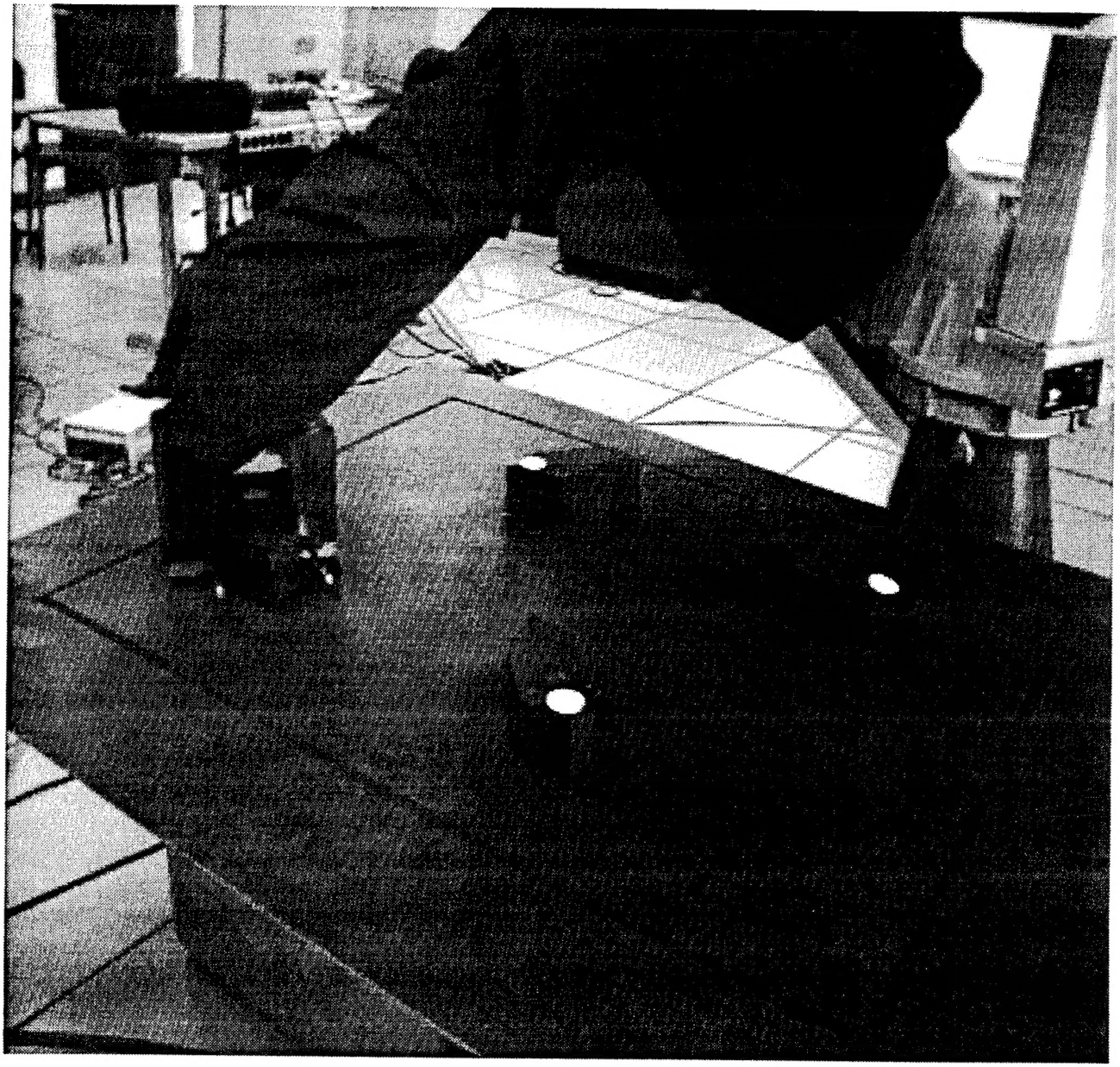

Figure 22: Shepherding setup

6 Conclusion

The real world shepherding application runs in our laboratory. The setup of the field, robot arm, and sheep can be seen in figure 22. The robot arm can keep one sheep on the field indefinitely, indicating that it can track and manipulate. It also performs equally well on two sheep, however mechanical difficulties sometime interfere. If the two sheep collide and stick the robot task is nonexistent. If the two sheep stick and move together the manipulator we have designed for the end of the robot arm does not allow it to turn the sheep back towards the center. Both of these events are unlikely with two sheep on the field but increase as we place more sheep on the field. The other difficulty with greater numbers of sheep is the relatively limited extent the robot can reach. This caused the size of the field to be quite small. The size of the field is approximate three feet by four feet, and each sheep is nine inches long and three and half inches wide. We have been able to confine four sheep on the field for a limited amount of time (about two saves each). The most limiting factor is the speed of the robot arm.

We have used the basic shepherding platform to create graduate student class projects. The implementation was robust enough to allow students to use it as a base and work with additional interesting variations. One group, *no helicopters* was not allowed to have an overhead camera or to survey the field from any height above several inches. This precluded the possibility of obtaining a global view of the world. A second group *clouds* had to handle multiple clouds throughout the field. This made for frequently obscured sheep and placed additional constraints upon quick acquisition and accurate tracking of the objects. As a final variation *wolves* we used a second robot arm to simulate a wolf entering the field. The original robot arm was equipped with a laser gun and had to "kill" the encroaching wolf by aiming at and hitting a sensor target placed on the second robot arm controlling the wolf.

We have described the implementation of shepherding, a real-world application combining vision, manipulation, and planning. This application has been successfully implemented in our hardware lab and can confine approximately four sheep in a three foot by four foot field. This project was part of a larger project of investigating the design issues of implementing support for parallel real-world applications. The implementation has allowed us to address areas where simulation was inadequate to do so. However, simulation allows us to address situations the real-world application does not. In the end we believe a better understanding of support and design and principles for SPARTAS will come from a combination of both techniques.

Acknowledgements

We gratefully acknowledge our advisor Christopher Brown. He provided motivation, guidance, and support during many phases of this project. We would also like to thank Tom LeBlanc and the 1993 CS400 class for exploring different variations and raising interesting issues.

References

[1] D.H. Ballard and C. M. Brown. Principles of animate vision. *cvgip*, 56(1):3–21, July 1992.

[2] Y. Bar-Shalom and T. E. Fortman. *Tracking and Data Association.* Academic Press, 1988.

[3] John Lloyd and Vincent Hayward. *RCCL - RCI System Overview.* McGill Research Centre for Intelligent Machines, McGill University, Montreal, Quebec, Canada, July 1992.

[4] Robert W. Wisniewski and Christopher M. Brown. Ephor, a run-time environment for parallel intelligent applications. In *Proceedings of The IEEE Workshop on Parallel and Distributed Real-Time Systems*, pages 51–60, Newport Beach, California, April 13-15, 1993.

[5] Robert W. Wisniewski and Christopher M. Brown. An argument for a runtime layer in sparta design. In *Proceedings of The 11th IEEE Workshop on Real-Time Operating Systems and Software*, pages 91–95, Seattle Wahington, May 18-19 1994.